A Prototype Implementation of MPI for SMARTMAP

Ron Brightwell

Sandia National Laboratories*
Scalable System Software Department
Albuquerque, NM USA
rbbrigh@sandia.gov

Abstract. Recently the Catamount lightweight kernel was extended to support direct access shared memory between processes running on the same compute node. This extension, called SMARTMAP, allows each process read/write access to another process' memory by extending the virtual address mapping. Simple virtual address bit manipulation can be used to access the same virtual address in a different process' address space. This paper describes a prototype implementation of MPI that uses SMARTMAP for intra-node message passing. SMARTMAP has several advantages over POSIX shared memory techniques for implementing MPI. We present performance results comparing MPI using SMARTMAP to the existing MPI transport layer on a quad-core Cray XT platform.

1 Introduction

Catamount [1] is a third-generation lightweight kernel developed by Sandia National Laboratories and Cray, Inc., as part of the Sandia/Cray Red Storm project [2]. Red Storm is the prototype of the Cray XT series of massively parallel machines. Recently, Catamount was enhanced using a technique called SMARTMAP – Simple Memory of Address Region Tables for Multi-core Aware Programming. SMARTMAP allows the processes running on a compute node as part of the same parallel job to efficiently read and write each other's memory. Unlike POSIX shared memory, SMARTMAP allows a process to access another process' memory by simply manipulating a few bits in a virtual address. This mechanism has several advantages for efficiently implementing MPI for intra-node communication.

We have developed a prototype MPI implementation using Open MPI that is able to use SMARTMAP for intra-node communication. Initial performance results show that SMARTMAP is able to achieve significant improvement for

* Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

intra-node point-to-point and collective communication operations. The following section describes the advantages of SMARTMAP compared to existing approaches for intra-node MPI. Section 3 provides a detailed description of the implementation of MPI communication using SMARTMAP, which is followed by a performance comparison between SMARTMAP and the existing transport for Red Storm. Relevant conclusions and an outline of future work are presented in Section 5.

2 Background

SMARTMAP takes advantage of the fact that Catamount only uses a single entry in the top-level page table mapping structure (PML4) on each core of a multi-core AMD Opteron processor. Each PML4 slot covers 39 bits of address space, or 512 GB of memory. Normally, Catamount only uses the first entry covering physical addresses in the range `0x0` to `0x007FFFFFFFFF`. The Opteron supports a 48-bit address space, so there are 512 entries in the PML4.

Each core writes the pointer to its PML4 table into an array at core 0 startup. Each time the kernel enters the routine to start a new context, the kernel copies all of the PML4 entries from every core into every other core. This allows every process on a node to see every other process' view of the virtual memory at an easily computed offset in its own virtual address space. The following routine can be used by a process to manipulate a "local" virtual address into a "remote" virtual address on a different core:

```
static inline void *remote_address( unsigned core, void *vaddr )
{
  uintptr_t addr = ((uintptr_t) vaddr) & ( (1UL<<39) - 1);
  addr |= ((uintptr_t) (core+1)) << 39;
  return (void*) addr;
}
```

SMARTMAP also takes advantage of Catamount's physically contiguous address space mapping and the fact that the address mappings are static. Unlike traditional UNIX-based operating systems, Catamount determines the mapping from virtual to physical addresses when a process is created and the mapping is never changed. Because each process from the same executable will have the same virtual address mapping, the location of variables with global scope will be identical across all of the processes – both on node and off node.

There is much previous work on using shared memory for intra-node MPI communications [3–5]. The traditional approach is to use a POSIX shared memory to allocate a region of memory that is shared between communication processes on a node. This memory is divided up among the processes and message queues are built inside the region. In order to send a message, the sender copies data into the shared region and the receiver copies it out. Other approaches that use only a single copy have been implemented and studied. One such implementation is a Linux kernel module that handles re-mapping of user memory pages into kernel space so that the operating system can do a single memory copy between processes [6]. Another approach is to have an intelligent or programmable network interface perform a single copy between processes on the same node. A comprehensive analysis of the different approaches for intra-node MPI communication was presented in [7].

Both of these approaches are currently used for the MPI implementation on Red Storm using the Portals data movement layer [8]. There are two different implementations of Portals available on Red Storm. The default implementation interrupts the operating system to service the network. For intra-node transfers, the operating system simply copies data between the processes. This approach is much like the Linux kernel module, except that Catamount's static memory mapping avoid having to do any re-mapping of pages. For larger messages, it is more efficient for the OS to use the SeaStar [9] network interface to perform the copy. In the second implementation of Portals, all network processing is performed on the SeaStar. Therefore, all intra-node transfers must go through the network interface.

The simplified memory model of Catamount means that there is no registration overhead, no system call, and no setup or teardown time necessary for one process to directly move data into another process. In addition, there is no serialization of processes through the operating system. Processes are free to move data without any OS involvement. Relative to using an intelligent network adapter, there is no need to have data traverse a I/O bus and there is no serialization of requests through the network interface. There is no synchronization mechanism needed to transfer large messages. Our current implementation has a single protocol for moving data. Since data only moves when both sender and receiver have initiated communication, there are no unexpected messages and no protocols needed to distinguish between the various MPI point-to-point send modes. We are able to achieve low latency, high throughput, significantly increased small message rate, overlap of computation and communication, and we are able to support both the active and passive MPI-2 one-sided functionality. Our approach is much simpler in term of resource allocation and management. For example, we are not constrained by a shared resource, such as how to best divide up shared memory regions between control and data.

In addition to these benefits, SMARTMAP allows for additional capabilities that existing approaches do not. For example, non-contiguous data transfers can simply be copied directly from sender to receiver with no intermediate copies or packing/unpacking. Collective operations can operate directly on the buffers involved in the communication. In particular, reduction operations can operate directly on the buffer in-place at the root of the operation. Using MPI_IN_PLACE, copying can be avoided altogether.

3 Prototype Implementation

Catamount supports multiple cores by running in virtual node mode, where each core is treated as a node that runs a process in the parallel job. The memory on a single physical node is divided evenly among the available cores. When a process is started, the parallel runtime system is responsible for setting the following values in the process' address space:

- number of processes in the job (`_my_nnodes`)
- global rank of the process (`_my_rank`)
- number of active cores on the node (`_my_vnm_degree`)
- core rank on the node (`_my_core`)

We then use these values to determine a global rank to core rank mapping and a core rank to global rank mapping. First, we allocate an array for the global rank to core rank mapping and initialize all entries to -1. We then allocate an array for the core rank to global rank mapping. We take advantage of the SMARTMAP capability to determine the core rank to global rank mapping. Each core loops from 0 to `_my_vnm_degree`. If the loop variable is equal to the local core's rank, it fills in the core rank to global rank mapping with its global rank. If the loop variable is not the process' core rank, it accesses the `_my_rank` value on the other core and fills in the array with this value. At this point, we can use this array to set the corresponding values in the global rank to core rank array to the appropriate values.

Each process has a single queue for posted receives and a queue per core for posted sends. These queues are doubly-linked lists. Each queue element contains: buffer address, local source rank, buffer length, context id, tag, request address, and completion flag.

In order to send a message to another core on the same node, we first check the global rank to core rank mapping to discover whether the destination rank is a local process. If it is, we check the destination core to see if the message is being sent to the sending process. If so, we traverse the posted receive queue looking for a match. If the message is destined for a different core on the same node, we simply pop a queue element off of a stack, fill in the contents based on the send request, and enqueue it on the send queue for that particular core. If the send is blocking, we then call into the progress routine, which is described in detail below.

When posting a receive, we check to see if the posted receive queue is currently empty. If it is, we can check for a match right away. In order to check for a match, we get the address of our core's send queue in our address space. Because this virtual address is identical across all of the processes on the node, we can easily use SMARTMAP to get the address of this queue in the other process' address space. If the source of the receive is specified, we get the address of our send queue in the other process' address space and proceed to traverse this queue looking for a matching send queue entry. Because the pointers are local to the destination process, every `next` pointer needs to be converted to a remote pointer to traverse the queue. When a match is found, the start address of the send buffer is converted to a remote pointer and the send buffer is copied from the sending process' memory into the local receiving process' memory. We then set the completion flag of the send queue element in the sending process' memory. If the source of the receive is MPI_ANY_SOURCE, we simply traverse our send queue in all of the other process' address space looking for match. If our posted receive queue is not empty, we simply pop a queue element off of a stack, fill in the contents, and enqueue it. We then call into the progress function to see if any outstanding requests have been or can be completed. In order to probe for an incoming messages, we do the same steps as posting a receive; however, when a match is found, we simply fill in the appropriate status information.

The progress function first checks to see if any outstanding posted receives can be completed. It does this in the same way as was just described in the previous paragraph – traversing our send queue in the other process' address space looking for match. After traversing the posted receive queue, the progress function traverses all of the send queues to see if any of the outstanding sends have been marked as completed by the other cores. Any send queue elements that have been marked as completed by the receiving core are dequeued and any cleanup routine for the request is run.

In this strategy, all queues are managed by the local core, so there are no race conditions with enqueueing and dequeueing elments. Receives are completed by the local core, copying any data from the remote core. Sends are completed by the remote core and dequeued by the local core.

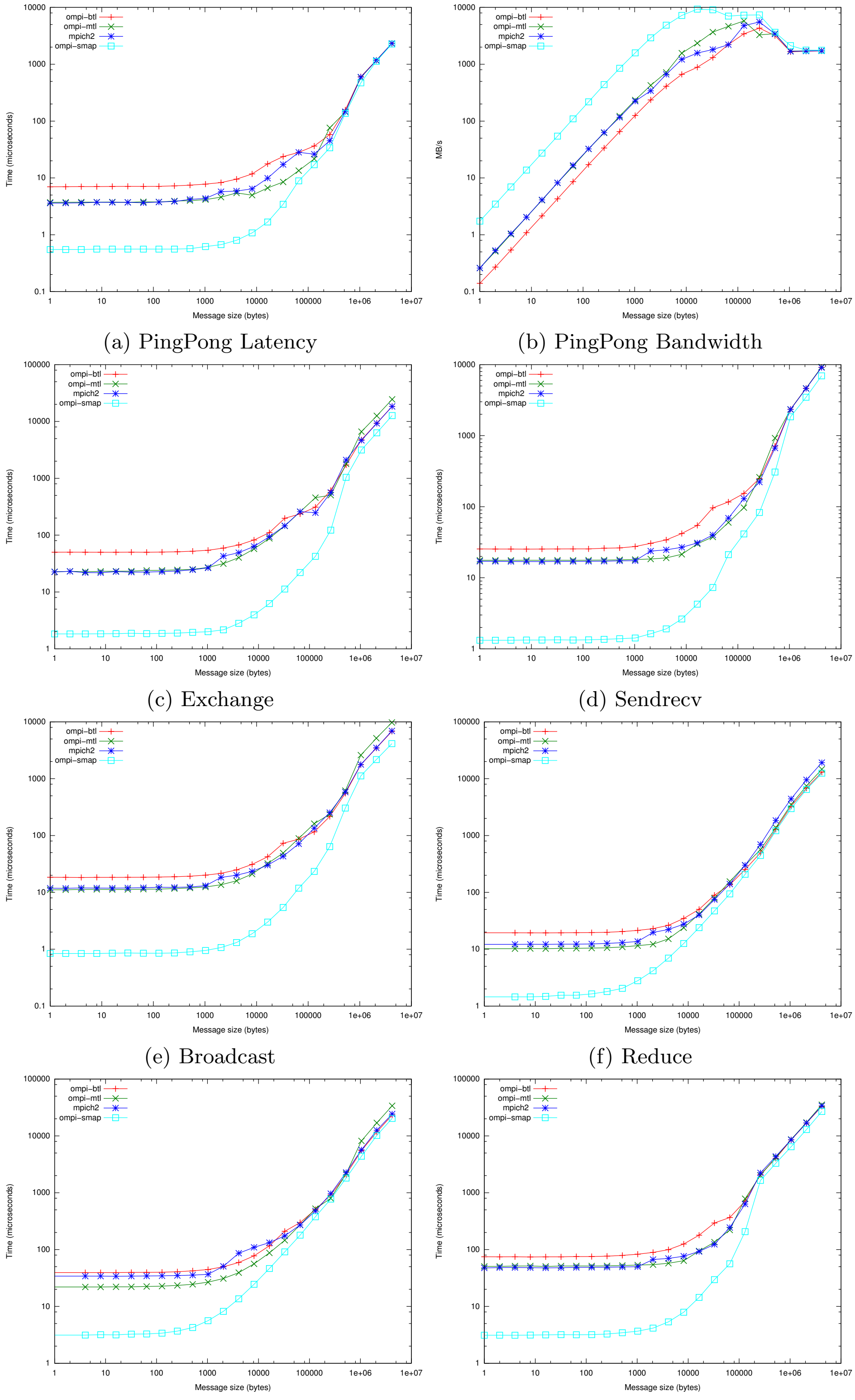
4 Performance Evaluation

The platform used to gather our performance results is a Red Storm development system that contains 2.2 GHz quad-core Opterons. Our prototype implementation was done using Open MPI. Open MPI already had support for Portals on the Cray XT, using either a path that does matching inside the MPI library (BTL) or one that does matching inside Portals (MTL). See [10] for a complete discussion of these two approaches. We also compare results to the Cray MPI implementation, which is an modified version of MPICH2.

We used the Intel MPI Benchmark Suite (IMB) version 2.3 to measure point-to-point and collective communication performance. In order to characterize small message rate, we used the Ohio State message rate benchmark that has been modified by PathScale (now Qlogic).

Since we are interested in intra-node communication, all of our results are from a single quad-core node. We limited our measurements to the interrupt-driven version of Portals because it is more efficient at intra-node transfers. The ability to have the operating system perform a copy between processes is more efficient than having the SeaStar adapter perform the copy. Due to limitations of the SeaStar, send operations must go through the OS, so in addition to serializing requests through a slower network interface, requests must also be serialized through the OS.

Figures 1(a) and 1(b) show ping-pong latency and bandwidth respectively. SMARTMAP is able to achieve a zero-byte latency of 520 ns, while Cray's MPI achieves 2.98 μs. SMARTMAP's bandwidth peaks at more than 9.3 GB/s, while the MTL in Open MPI is able to achieve nearly 5.8 GB/s. This difference is



(a) PingPong Latency

(b) PingPong Bandwidth

(c) Exchange

(d) Sendrecv

(e) Broadcast

(f) Reduce

(g) Allreduce

(h) Alltoall

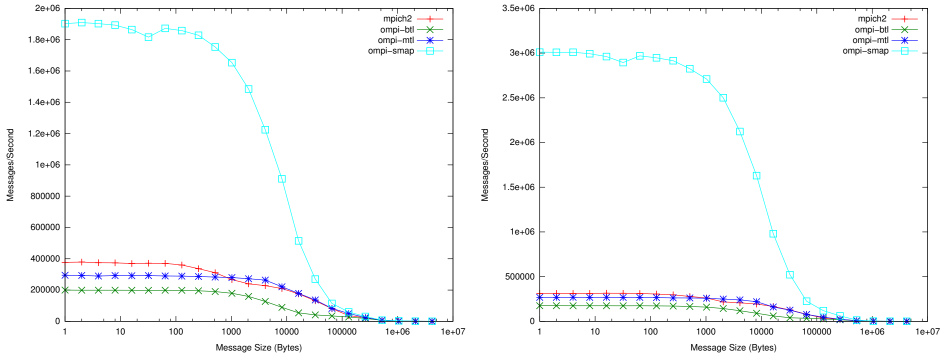
Fig. 1: IMB Results

likely due to the extra serialization through the OS and overhead incurred by having the OS perform the memory copies.

Figures 1(c) and 1(d) show the performance of four-process exchange and sendrecv operations respectively. These results again show that the impact of serialization worsens as all four cores are attempting to exchange message simultaneously.

Figures 1(e) and 1(f) show the performance of four-process broadcast and reduction collective operations. The SMARTMAP broadcast is less than 1 μs out to a 1024-byte message size, at which point the others are all more than 12 μs. The reduce operation shows a similar performance differential.

We conclude IMB performance results with allreduce and alltoall performance in Figures 1(g) and 1(h). These results again demonstrate the effectiveness of using shared memory when all processes are communicating simultaneously. The alltoall results are particularly dramatic, especially at larger message sizes.



(a) One pair

(b) Two pairs

Fig. 2: Message Rate

Figures 2(a) and 2(b) show message rates for two process and four processes respectively. SMARTMAP is able to achieve nearly 2 million messages per second for two processes and over 3 million messages per second when two pairs of processes are exchanging messages. Performance for the others actually decreases when increasing from two to four processes.

5 Conclusion

The SMARTMAP capability in the Catamount lightweight kernel is able to deliver significant performance improvements for intra-node MPI point-to-point and collective operations. It is able to dramatically outperform the current approaches for intra-node MPI data movement.

There is much work left to do to fully utilize the SMARTMAP capability for MPI. First, because the Portals data movement layer encapsulates the MPI posted receive queue, the complexity of handling MPI_ANY_SOURCE receives is significantly increased. In most other networks, the MPI posted receive queue exists inside the MPI library, so the atomicity required to handle a non-specific receive is straightforward. Because Portals contains the posted receive queue for network transfers (either in the OS or in the network interface) and the MPI library contains the posted receive queue for intra-node transfers, there is no mechanism by which atomicity can be enforced. There are two possible choices for handling MPI_ANY_SOURCE receives for communicators that are not either completely on-node or completely off-node. If the request cannot be satisfied by any current in-progress communication, all current shared memory transfers need to queued as unexpected messages and the peer processes need to be informed that all subsequent messages should use Portals rather than shared memory. Alternatively, the implementation could modify the Portals implementation so that all matching occurs inside the MPI library, rather than inside Portals. Cray's MPI and the Open MPI BTL already support this mode of operation. Either way – by putting the MPI posted receive queue completely in the network or completely inside MPI – great care must be taken to avoid race conditions and to maintain MPI ordering semantics, both in order to satisfy the non-specific receive and to possibly switch back after it has been completed. We are currently adapting both the Open MPI Portals MTL and the shared memory BTL to be able to use SMARTMAP. This will allow us to support both intra- and inter-node communication and will allow for direct comparisons between SMARTMAP and the POSIX shared memory approach.

From a complexity standpoint, it is much easier to enhance MPI collective operations to use the SMARTMAP capability. It is not necessary to handle non-specific receive operations or non-blocking operations. We can also make use of direct shared memory for certain operations, like barrier, when incrementing a shared counter may be more efficient than exchanging messages. We are currently implementing a collective module in Open MPI to use SMARTMAP directly and hope to take advantage of the existing hierarchical collective module to make use of it.

We would also like to use SMARTMAP to handle on-node non-contiguous data transfers with no intermediate buffering, and there is an opportunity to enhance the on-node MPI-2 one-sided operations using SMARTMAP as well.

6 Acknowledgments

Trammell Hudson is responsible for the implementation of SMARTMAP in Catamount, which was an outcome of discussions between the author and Kevin Pedretti. This work would also not have been possible without John Van Dyke, who is responsible for implementing virtual node mode support in Catamount. The author would also like to thank Sue Kelly and the Cray support staff at Sandia for assistance with the Red Storm development systems.

References

1. Kelly, S.M., Brightwell, R.: Software architecture of the light weight kernel, Catamount. In: Proceedings of the 2005 Cray User Group Annual Technical Conference. (2005)
2. Camp, W.J., Tomkins, J.L.: Thor's hammer: The first version of the Red Storm MPP architecture. In: In Proceedings of the SC 2002 Conference on High Performance Networking and Computing, Baltimore, MD (2002)
3. Buntinas, D., Mercier, G., Gropp, W.: Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem. Parallel Computing **33**(9) (2007) 634–644
4. Buntinas, D., Mercier, G., Gropp, W.: Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem. In: Proceedings of the 2006 European PVM/MPI Users' Group Meeting. (2006)
5. Buntinas, D., Mercier, G., Gropp, W.: Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In: Proceedings of the 2006 International Symposium on Cluster Computing and the Grid. (2006)
6. Jin, H.W., Sur, S., Chai, L., Panda, D.K.: Limic: Support for high-performance MPI intra-node communication on Linux. In: Proceedings of the 2005 Cluster International Conference on Parallel Processing. (2005)
7. Buntinas, D., Mercier, G., Gropp, W.: Data transfers between processes in an smp system: Performance study and application to mpi. In: Proceedings of the 2006 International Conference on Parallel Processing. (2006)
8. Brightwell, R., Hudson, T., Pedretti, K., Riesen, R., Underwood, K.: Implementation and performance of Portals 3.3 on the Cray XT3. In: Proceedings of the 2005 IEEE International Conference on Cluster Computing. (2005)
9. Brightwell, R., Hudson, T., Pedretti, K.T., Underwood, K.D.: SeaStar interconnect: Balanced bandwidth for scalable performance. IEEE Micro **26**(3) (2006)
10. Graham, R.L., Brightwell, R., Barrett, B., Bosilca, G., Pjesivac-Grbovic, J.: An evaluation of Open MPI's matching transport layer on the Cray XT. In: Proceedings of the 14th European PVM/MPI Users' Group Conference. (2007)